Lezione 6

Passaggio di parametri per riferimento: puntatori
Allocazione dinamica array

Informatica, Corso B - D. Tamascelli

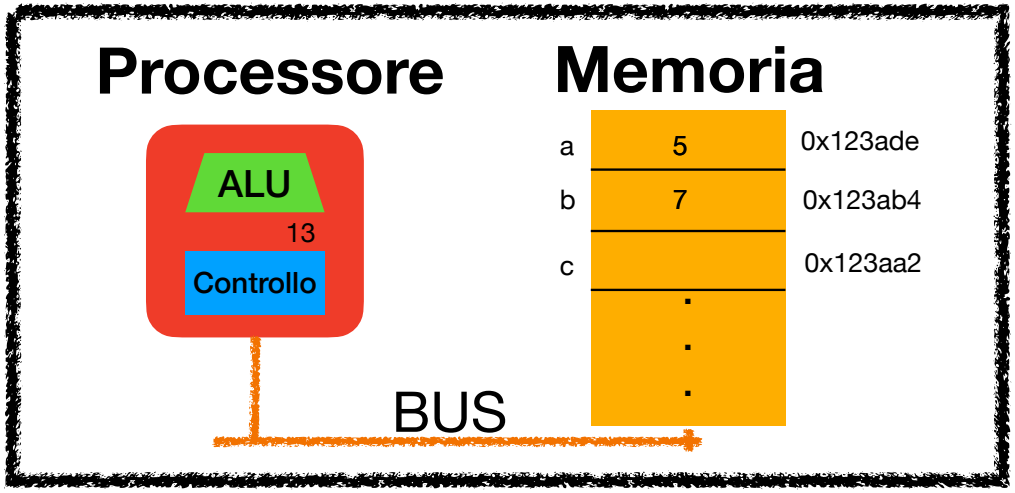
Puntatori e passaggio parametri

Variabile: definizione

Una variabile (di programma) è:

- Il nome mnemonico associato univocamente ad una "zona" (locazione) di memoria
- contenente informazione, ovvero una sequenza di bit
- di lunghezza e codifica determinati da un tipo

Abbiamo già visto il concetto di variabile in azione nell'esempio della somma



Tipi indirizzo

- variabile tipo `char`: contiene informazione di tipo carattere (8 bit, 1 byte)
- variabile tipo `int`: contiene informazione di tipo intero (32 bit, 4 byte)
- variabile tipo `float`: contiene informazione di tipo razionale (sp) (32 bit, 4 byte)
- variabile tipo `double`: contiene informazione di tipo razionale (dp) (64 bit, 8 byte)

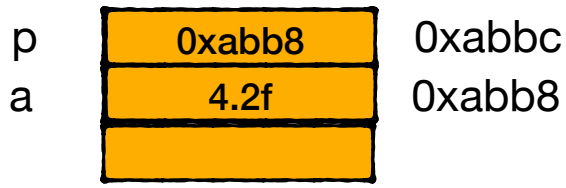
- variabile tipo `char *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un char.
- variabile tipo `int *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un int.
- variabile tipo `float *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un float.
- variabile tipo `double *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un double.

Tipi indirizzo

`T *` : è un tipo di dato

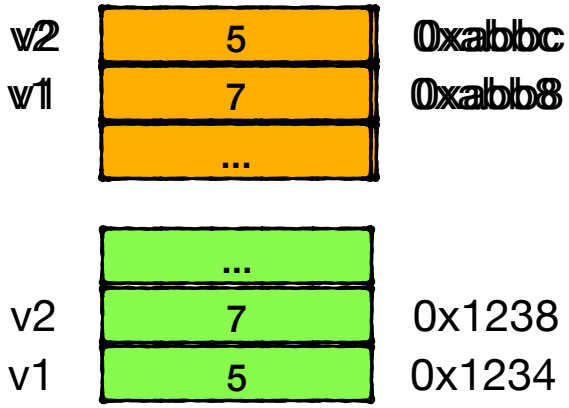
`T * p`: è una variabile **di nome** p **di tipo** `T *` capace di contenere l'indirizzo di una cella di memoria contenente un dato di tipo `T`.

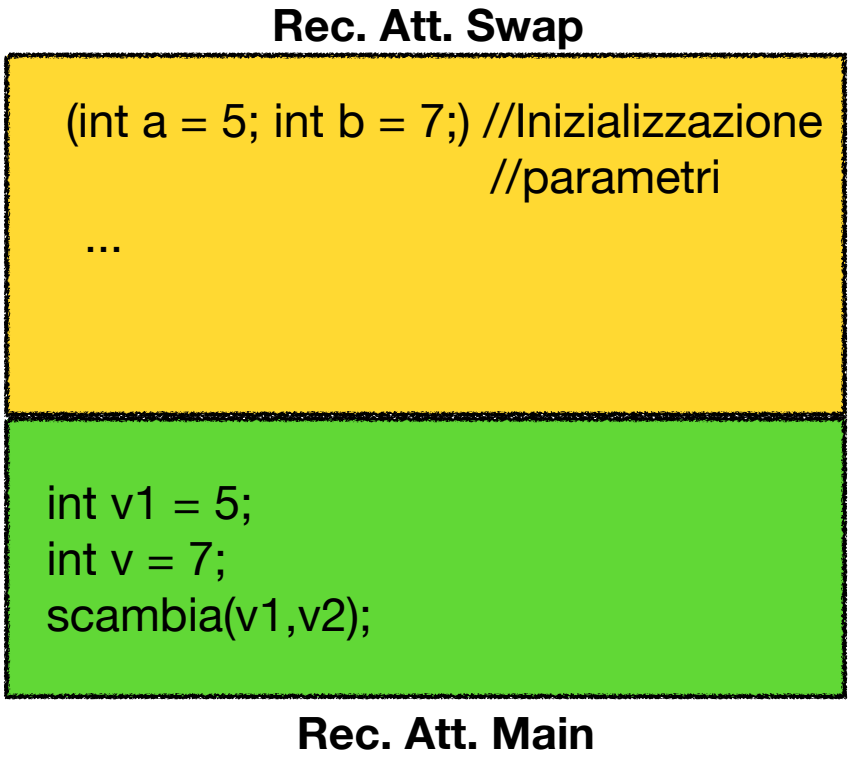
```
float a = 3.2f;

float * p;

p = &a; //Assegna a p l'indirizzo di a

*p = 4.2f; //Scrivi nell'indirizzo contenuto in p  il valore 4.2f
```



Problema: scambia (swap)

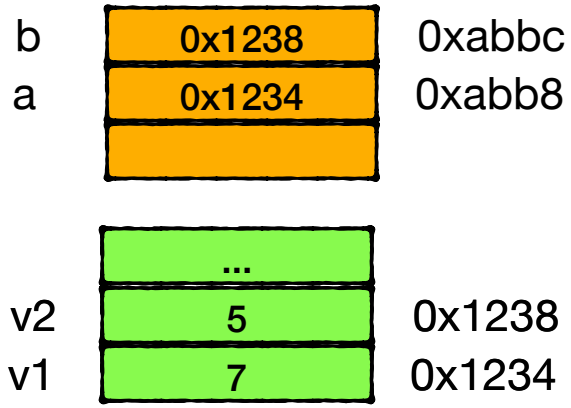
```
int main(){

int v1=5;

int v2=7;

scambia(v1,v2);

}

void scambia(int a, int b){
int appo;
appo =a;
a=b;
b=appo;
}
```

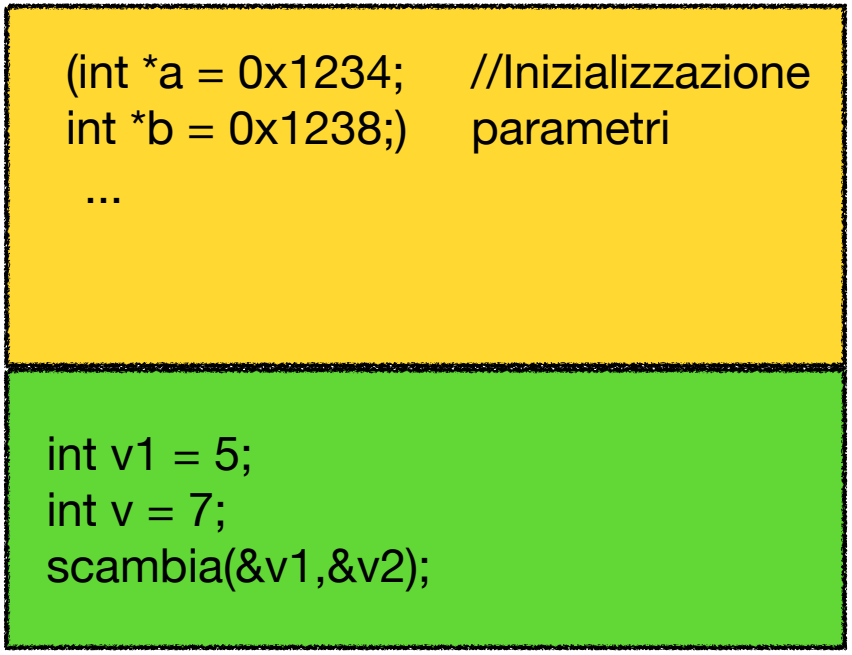




Swap per side effect

```
int main(){

int v1=5;

int v2=7;

scambia(&v1,&v2);

}

void scambia(int *a, int *b){
int appo;
appo = *a;
*a = *b;
*b = appo;
}
```





Swap per side effect (2)

```
void scambia(int *a, int *b){
int appo;
appo = *a;
*a = *b;
*b = appo;
}
```

- *a: all'indirizzo contenuto in a (***a** compare come **RHS**),
- leggi sizeof(int) byte
- da interpretare come intero
- e assegna (scrivi in) appo.

- *b: all'indirizzo contenuto in b (***b** compare come **RHS**),
- leggi sizeof(int) byte
- da interpretare come intero
- e assegna alla cella di memoria di indirizzo contenuto in a.

(***a** appare come **LHS**)

***: operatore unario di deferenziazione**

***v: alias della variabile all'indirizzo contenuto in v, di cui conosciamo anche il tipo.**

- assegna il valore contenuto in appo (RHS)
- alla cella di memoria di indirizzo contenuto in b

(***b** appare come **LHS**)

Quindi

- Parametri formali di tipo puntatore possono essere usati per permettere ad una funzione di modificare **variabili locali** di altre funzioni/procedure.
- Quindi permettono di "esportare" dalle funzioni, se necessario, più di un valore.
- "Esportare": modificare lo stato di memoria (contenuto): side effect.

Allocazione dinamica array

La dimensione "giusta"

Riflessioni

- Spesso il numero dei dati che il nostro programma si troverà a elaborare NON è noto a compile-time, ovvero quando il programma viene scritto

- Una soluzione è usare array di dimensione "abbondante" per il problema.

- Ovviamente con questo approccio non possiamo andare molto lontano:

  ✳Se i dati dovessero eccedere la disponibilità, dovremmo riscrivere il programma, allargando i vettori lì dichiarati.

  ✳Chiaramente la strategia di allocare vettori "ENORMI", per evitare di modificare il programma, è assurda: occuperemmo memoria per nulla.

- Al momento, però, non abbiamo strade alternative...

- ...e l'uso dei file ha messo in chiaro che, tipicamente, il numero dei dati che il nostro programma si troverà a manipolare NON è, tipicamente, noto a compile time.

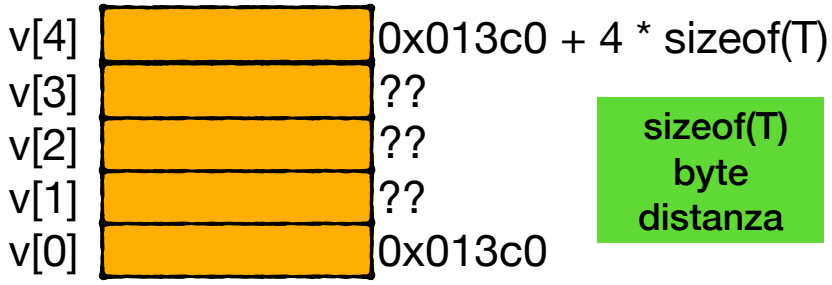
Quindi...?

Un array è:

- una n-upla ordinata
- di elementi dello stesso tipo
- allocata in una zona di memoria senza soluzione di continuità
- ciascuna componente del quale è quindi accessibile se sono noti:

✓L'indirizzo di partenza dell'array

✓La dimensione (in byte) di ciascuna componente



- Dato un vettore di 5 elementi di tipo T

- se indichiamo con &v[0] la base (punto di partenza dell'array), ovvero l'indirizzo del primo elemento dell'array:

$$v[i] \longleftarrow base + i * sizeof(T)$$

Se solo....

Se:

- fossimo capaci di riservare, durante l'esecuzione del programma, una zona di memoria...
- ...senza soluzione di continuità....
- ...della dimensione giusta per contenere n*sizeof(T) elementi di tipo T
- (dove n è determinato durante l'esecuzione del programma)
- ...e di registrare da qualche parte l'indirizzo di memoria dove la zona inizia...

Allora:

Saremmo in grado di creare i nostri contenitori di informazione array, della dimensione necessaria, durante l'esecuzione del programma!

In effetti....

- fossimo capaci di riservare, durante l'esecuzione del programma, una zona di memoria...
- ...senza soluzione di continuità....
- ...della dimensione giusta per contenere n elementi di tipo T (n * sizeof(T)
- (dove n è determinato durante l'esecuzione del programma)

Istruzione (C++)

`new T[n]`

`new T[n]`: riserva al programma una zona di memoria di dimensione `n*sizeof(T)` e restituisce l'**indirizzo di memoria** dove comincia l'area riservata.

- ...e di registrare da qualche parte l'indirizzo di inizio...

???

Tipi indirizzo

- variabile tipo `char`: contiene informazione di tipo carattere (8 bit, 1 byte)
- variabile tipo `int`: contiene informazione di tipo intero (32 bit, 4 byte)
- variabile tipo `float`: contiene informazione di tipo razionale (sp) (32 bit, 4 byte)
- variabile tipo `double`: contiene informazione di tipo razionale (dp) (64 bit, 8 byte)

- variabile tipo `char *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un char.
- variabile tipo `int *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un int.
- variabile tipo `float *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un float.
- variabile tipo `double *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un double.

Tipi indirizzo (2)

`T *` : è un tipo di dato

`T * p`: è una variabile di tipo `T *` capace di contenere l'indirizzo di una cella di memoria contenente un dato di tipo `T`.

Se qualcuno si stesse chiedendo a che cosa serve sapere il tipo del dato all'indirizzo CONTENUTO in p....

.....CI ARRIVIAMO!

```
float * p;

int n;

cin >> n;

p = new float[n];
```

Et voilá: abbiamo creato un'area di memoria capace di contenere n float, il cui **indirizzo è stato registrato in p**!

Allocazione dinamica di array

`float * p;`

- dichiariamo una variabile capace di contenere l'indirizzo di una variabile di tipo float

int n;

cin >> n;

- leggo da tastiera, o determino comunque a run-time, in numero dei dati

p = new float[n];

- uso il comando `new` per allocare in memoria
  - ✤una zona di dimensione giusta per contenere n*sizeof(T) elementi di tipo T
  - ✤restituendo l'indirizzo di memoria in cui questa area inizia.
  - ✤...e adesso abbiamo il contenitore di informazione (variabile) del tipo giusto per registrare questo valore.

E adesso?

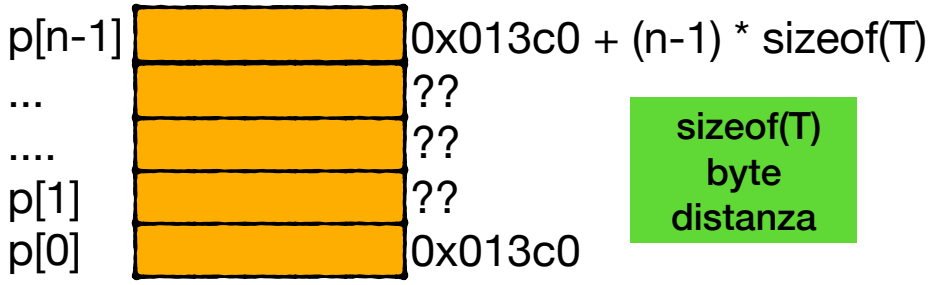
Allocazione dinamica di array (2)

Un array è:

- una n-upla ordinata
- di elementi dello stesso tipo
- allocata in una zona di memoria senza soluzione di continuità
- ciascuna componente del quale è quindi accessibile se sono noti:

✓L'indirizzo di partenza dell'array

✓La dimensione (in byte) di ciascuna componente



possiamo usare p come un vettore!!!!

- p **contiene** la base (punto di partenza) dell'array, ovvero l'**indirizzo** del primo elemento dell'array:

$$p[i] \longleftarrow base + i * sizeof(float)$$

Allocazione dinamica di array (3)

```
float *  p;

int n;

cin >> n;

p = new float[n];
```

Da questo momento in poi possiamo usare p come un normalissimo array:"

```
p[0] = 2.3;

p[1] = p[0] + 5.;
```

```
flusso_in >> appo;
while(!flusso_in.eof()){
   p[i] = appo;
   i++;
   flusso_in >> appo;
}
```

"With great power comes great responsibility"

Uncle Ben (Spider Man)

- La **memoria allocata dinamicamente rimane** riservata al programma anche quando la funzione che l'ha allocata muore.
  - Un array, o comunque una zona di memoria, allocata dinamicamente da un programma, **deve essere liberata dal programma stesso**.
- Altrimenti si rischia di mantenere occupata memoria inutilmente (**garbage**).

Quando è il programmatore a gestire esplicitamente l'allocazione della memoria TUTTO il CICLO di VITA delle ZONE ESPLICITAMENTE ALLOCATE è SOTTO LA RESPONSABILITÀ del PROGRAMMATORE

Deallocazione array dinamicamente allocati

```
float *  p;

int n;

cin >> n;

p = new float[n];
```

```
p[0] = 2.3;

p[1] = p[0] + 5.;
```

```
flusso_in >> appo;
while(!flusso_in.eof()){
   p[i] = appo;
   i++;
   flusso_in >> appo;
}
```

Finito di usare l'array

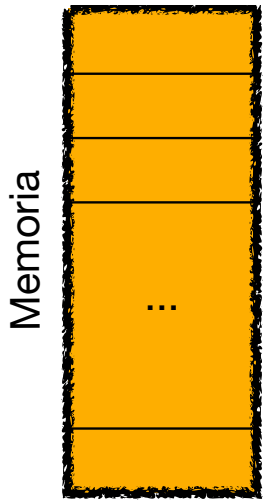
```
delete [] p;

p = NULL;
```

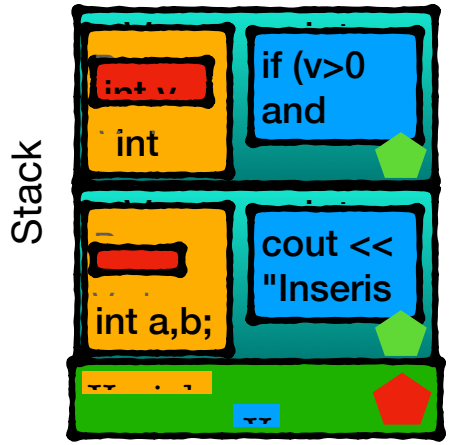
- delete [] p: libera la memoria
- p = NULL; : convenzione/igiene: p ora non si riferisce a nulla;
- successivi tentativi di accesso a p (per esempio p[3] = 1.f) portano a errori a runtime (il programma viene fatto terminare con qualche "parolaccia" dall'esecutore (segmentation fault...)

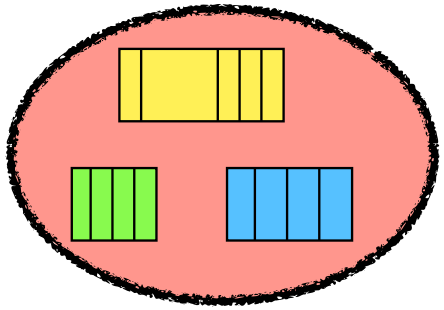
Stack vs Heap (Cultura generale)

Un programma in esecuzione ha accesso a zone di memoria organizzate in modo diverso:

- **Stack**: è l'area di memoria dove vengono allocati i record di attivazione di procedura. La gestione è lasciata al SO, che si occupa di caricare/scaricare i record secondo la politica già discussa
- **Heap (mucchio)**: è l'area di memoria dove vengono allocati i blocchi richiesti esplicitamente dal programma durante la sua esecuzione. La gestione è completamente nelle mani del programmatore







Altro uso: valore di ritorno

- Prendiamo la funzione:

```
float * f(int p);
```

- La funzione riceve quindi in ingresso un intero, che rappresenta la dimensione di un array da allocare dinamicamente.
- La funzione si occupa di allocare il vettore dinamicamente e di restituire l'indirizzo.

```
float *f(int p){

float *v = NULL;

v = new float[p];

return v;
}
```

```
float *vett;

int dim;

cin >> dim;

vett = f( dim);
```

Altro uso: caricamento file

- Prendiamo la funzione:

```
float * f(char nomefile[], int *p);
```

- La funzione riceve quindi in ingresso un nome di file e l'indirizzo di un intero.

```
float *f(char nomefile[], int *p){
float *v = NULL;
int conta = 0;
//Conta dati
v = new float[conta];
//Carica dati
*p =  conta; //Assegna alla variabile il cui indirizzo e` contenuto in p il valore di conta
return v; //Restituisci l'indirizzo dell'array dinamicamente allocato
}
```

Parametri puntatore

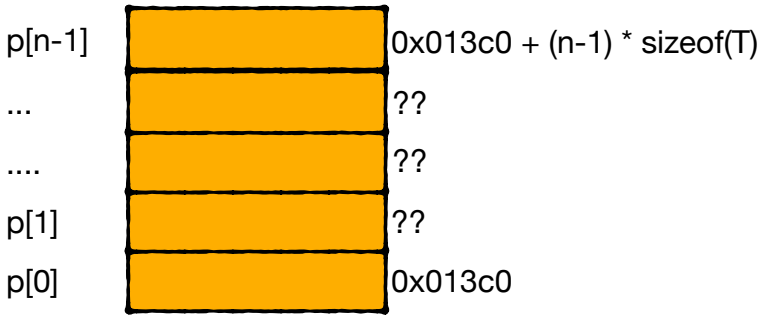
DON'T PANIC!!!!

- Hanno messo in luce un meccanismo particolare di passaggio di informazione tra funzioni
- Il meccanismo prevede di passare ad una funzione l'indirizzo di una variabile (zona di memoria) e di sfruttarlo per poter accedere a variabili NON locali della funzione
- Originariamente usato solo per variabili di tipo array (di qualsiasi tipo), ora può essere usato per "esportare" più valori di una funzione tramite side effects.

- By the way: ermettono di gestire l'allocazione dinamica della memoria

Ancora più attenzione

- Abbiamo visto che la ragione che consente a funzioni di modificare array passati come parametro è il fatto che alla funzione arriva l'indirizzo di inizio dell'array (e il tipo).




```
float media(float v[],int dim)  = float media (float * v, int dim)
```

- Un parametro formale array (float []) è quindi in realtà sempre stata una variabile di tipo puntatore a float (float *).